Code-Aware Cross-Program Transfer Hyperparameter Optimization

Zijia Wang, Xiangyu He, Kehan Chen, Chen Lin[*], Jinsong Su
School of Informatics, Xiamen University
Xiamen, Fujian, China
chenlin@xmu.edu.cn

Abstract

Hyperparameter tuning is an essential task in automatic machine learning and big data management. To accelerate tuning, many recent studies focus on augmenting BO, the primary hyperparameter tuning strategy, by transferring information from other tuning tasks. However, existing studies ignore program similarities in their transfer mechanism, thus they are sub-optimal in cross-program transfer when tuning tasks involve different programs. This paper proposes CaTHPO, a code-aware cross-program transfer hyperparameter optimization framework, which makes three improvements. (1) It learns code-aware program representation in a self-supervised manner to give an off-the-shelf estimate of program similarities. (2) It adjusts the surrogate and AF in BO based on program similarities, thus the hyperparameter search is guided by accumulated information across similar programs. (3) It presents a safe controller to dynamically prune undesirable sample points based on tuning experiences of similar programs. Extensive experiments on tuning various recommendation models and Spark applications have demonstrated that CatHPO can steadily obtain better and more robust hyperparameter performances within fewer samples than state-of-the-art competitors.

1 Introduction

Hyperparameter tuning is an important task in machine learning and big data management. For example, hyperparameters of Machine Learning (ML) models (e.g., learning rates and batch size) affect learning accuracy (Bergstra et al. 2011). Configuration knobs of Spark[1] (e.g., the memory size and number of cores for each executor) have a great impact on the execution time of a Spark application (Srinivasa and Muppalla 2015). Therefore, `HyperParameter Optimization` (HPO) (Thornton et al. 2013) has attracted much attention in both machine learning (He, Zhao, and Chu 2021) and database communities (Li, Zhou, and Cao 2021).

Bayesian Optimization (BO) (Jones, Schonlau, and Welch 1998) has been successfully applied in HPO. BO treats the hyperparameter performance as a black-box function which is expensive to evaluate. BO iteratively (1) updates a *probabilistic surrogate model* which measures the uncertainty of hyperparameter performance based on previous evaluations, and (2) computes an *acquisition function* based on the surrogate and sample the next hyperparameters to evaluate. Conventional BO suffers from the "cold-start" problem (Swersky, Snoek, and Adams 2013), i.e., since each *tuning task* searches from scratch, it needs to explore more samples (i.e., sample efficiency) and causes a large tuning overhead (i.e., time efficiency). Here a tuning task refers to hyperparameter tuning for a specific program (e.g., an ML model or a Spark application) on a certain dataset. To address the cold-start problem, a major and ongoing thrust of work has attempted to transfer information from other tuning tasks (i.e., source tasks), under the premise that hyperparameter performances of similar tuning tasks are likely to be similar (Bardenet et al. 2013).

However, defining similar tasks and their transfer mechanism remains an open problem. Some previous studies learn task similarities from evaluations of each task (Swersky, Snoek, and Adams 2013; Wistuba, Schilling, and Schmidt-Thieme 2016; Feurer, Letham, and Bakshy 2018; Soares and Brazdil 2000). For these methods, providing ample evaluations to accurately estimate task similarity is a hurdle. Other studies determine similar tasks based on manually defined meta-features of datasets (Yogatama and Mann 2014; Feurer, Springenberg, and Hutter 2015; Schilling, Wistuba, and Schmidt-Thieme 2016; Law et al. 2019). While these methods are useful in tuning a same ML model on different datasets, they are sub-optimal in *cross-program tuning*, e.g., tuning different ML models or Spark applications, because the hyperparameter performances are usually inconsistent for different programs. The nature of a program – the key element in a tuning task, has been largely ignored in existing studies.

This paper presents CaTHPO, **C**ode-**a**ware cross-program **T**ransfer **H**yper**P**arameter **O**ptimization. CaTHPO makes the following contributions. (1) Since the program code is its most distinguishing feature, a code-aware program representation module is proposed. This module is trained with self-supervised learning, so that it gives an off-the-shelf estimate of program similarities based on program representations. (2) The program similarities are used to adjust a Gaussian Process surrogate and a neural acquisition function network.

[*]Chen Lin is the corresponding author

[1]https://spark.apache.org/docs/latest/configuration.html

Thus, the hyperparameter search is guided by accumulated information across similar programs, and the tuning can be more sample efficient. (3) The program similarities are used in a safe controller to dynamically prune search space, based on bad hyperparameters of similar programs. Thus, undesirable hyperparameters will not be sampled, and the tuning can be more time efficient.

Extensive experiments are conducted on tuning various Spark applications and recommendation models. Experimental results show that, compared with state-of-the-art competitors, CaTHPO can steadily find better hyperparameters in shorter runs for different programs. Performances of CaTHPO's sampled hyperparameters are significantly more robust.

2 Related Work

Numerous studies have been proposed to increase BO's sample efficiency by transferring information from related tuning tasks. We briefly introduce their choices of surrogate models, acquisition functions and search paradigms.

Gaussian Process (GP) is the most commonly adopted surrogate model (Jones, Schonlau, and Welch 1998). Some existing studies use a single *global GP* to represent multiple tuning tasks, including SCoT (Bardenet et al. 2013), MTGP (Swersky, Snoek, and Adams 2013), wsKG (Soares and Brazdil 2000), and DistBO (Law et al. 2019). However, it incurs a prohibitive cost of computing the GP if there are many tuning tasks. Efforts to circumvent the scalability limitation fall into three categories. They either (1) build a hierarchical global GP where each level is trained on levels below, e.g., SHGP (Tighineanu et al. 2022) and HyperTune (Golovin et al. 2017); or (2) build separate *local GPs* for each tuning task. These local GPs can be isolated (Wistuba, Schilling, and Schmidt-Thieme 2018) or related. For example, the kernel of the target GP can look at points from nearest neighbor source tasks (Yogatama and Mann 2014). MisoKG (Poloczek, Wang, and Frazier 2017) assumes that the kernels of source tasks are approximating the target task with variable bias. POE (Schilling, Wistuba, and Schmidt-Thieme 2016) directly uses product of Gaussian experts trained on source tasks as the target surrogate. TSTR (Wistuba, Schilling, and Schmidt-Thieme 2016) and RGPE (Feurer, Letham, and Bakshy 2018) both build an ensemble target surrogate by linearly combining source GPs. Env-GP (Joy et al. 2016) stretches the target function noise to fit the evaluation difference between source and target tasks. Diff-GP (Shilton et al. 2017) improves over Env-GP by using bias-corrected source evaluations to update the target GP. (3) The GP can be replaced by a neural network. For example, BOHAMIANN (Springenberg et al. 2016) uses a Bayesian neural network whose input includes a task-specific embedding vector. ABLR (Perrone et al. 2018) builds a Bayesian linear regression surrogate for each tuning task and requires all surrogates to share an underlying basis expansion learned from a feedforward neural network. FSBO (Wistuba and Grabocka 2021) builds a shared deep surrogate for all tasks and adopts a deep kernel to learn the surrogate's task-independent parameters.

Information transfer can also be encoded in the Acquisition Function (AF). Expected Improvement (EI) (Jones, Schonlau, and Welch 1998) has been adopted by a large body of literature. TAF (Wistuba, Schilling, and Schmidt-Thieme 2018) defines a new AF as a weighted superposition of EI on the target task and the predicted improvements on source tasks. Some previous studies consider incremental gain per unit cost, including misoKG (Poloczek, Wang, and Frazier 2017) and MTGP (Swersky, Snoek, and Adams 2013). Instead, Monte Carlo estimates of the AF is adopted in (Snoek, Larochelle, and Adams 2012). Other than handcrafted AFs, recent studies have turned to *neural AF*. For example, MetaBO (Volpp et al. 2020) and FSAF (Hsieh, Hsieh, and Liu 2021) both train a neural network which represents AF across tasks via reinforcement learning.

Finally, A few studies have investigated transfer in the initialization and search space of BO. For example, a static sequence of hyperparameters is optimized across tasks in (Wistuba, Schilling, and Schmidt-Thieme 2015). MI-SMBO (Feurer, Springenberg, and Hutter 2015) initializes search on target task with best configurations on similar source tasks. BLB (Anderson et al. 2017) proposes to subsample in parallel based on Bag of Little Bootstraps. Low-volume ellipsoid search space is found to be superior than traditional rectangular boxes in (Perrone and Shen 2019).

Remarks. (1) Task similarity plays a key role in transfering information among tuning tasks. Previous studies capture task similarities by a fixed relative importance of new and old tasks (Golovin et al. 2017), manually defined meta-features of datasets (e.g., descriptive statistics of the datasets) (Yogatama and Mann 2014; Feurer, Springenberg, and Hutter 2015; Schilling, Wistuba, and Schmidt-Thieme 2016; Law et al. 2019), or evaluation similarities (e.g., pairwise ranking of evaluations in each task) (Swersky, Snoek, and Adams 2013; Wistuba, Schilling, and Schmidt-Thieme 2016; Feurer, Letham, and Bakshy 2018; Soares and Brazdil 2000). To the best of our knowledge, all existing studies are empirically verified on tuning the same ML model on different datasets. We argue that their assumption is too strong for cross-program transfer, e.g., good performing hyperparameters for an ML model A do not necessarily perform well for a dissimilar model B. Thus, we present a *novel transfer framework based on learned program similarities*, and we conduct *thorough experimental studies on cross-program tuning*. (2) Most existing studies focus on one or two components of the BO. Our code-aware program representation is incorporated in the *whole process of BO*, i.e., adjusting the surrogate and AF, as well as pruning the search space. Thus the sample efficiency and time efficiency of BO can be simultaneously enhanced by transfer learning. (3) The proposed CaTHPO adopts separate local GPs and a neural AF. Therefore, it *intensifies the AF's capability by learning a shared AF from multiple tasks, while maintaining a good scalability on the GP*.

3 Model

We are interested in cross-program hyperparameter tuning. The setting can be formalized as follows. Given T tasks,

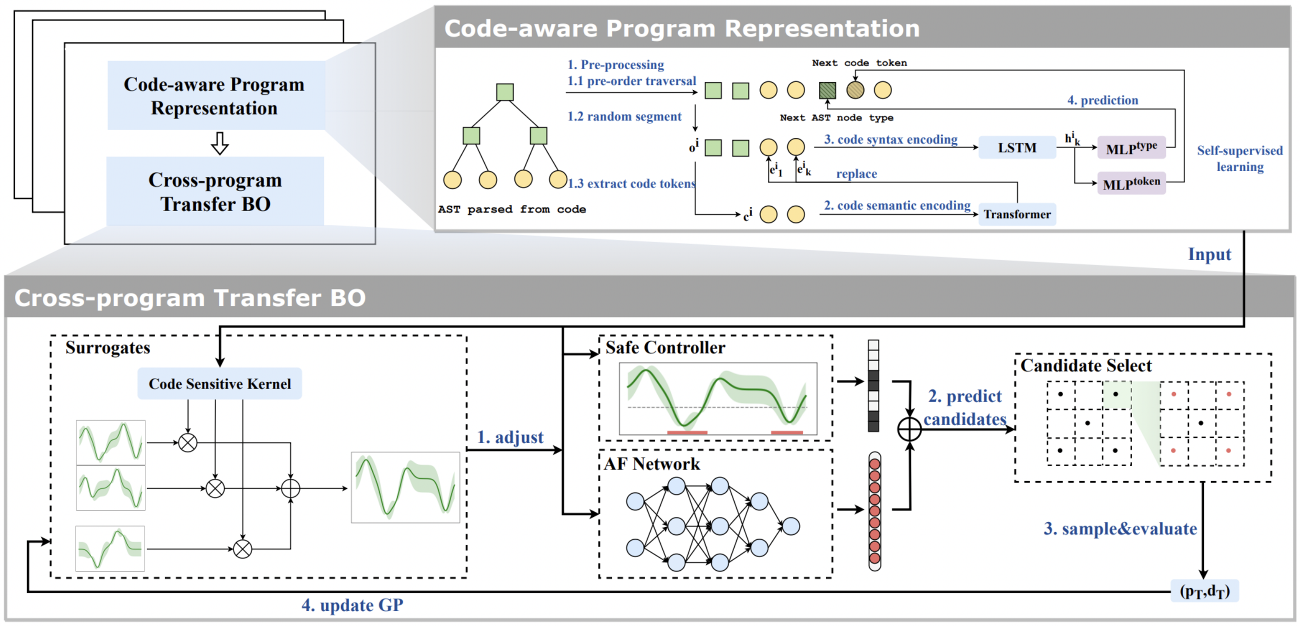

Figure 1: Framework overview of CaTHPO

where each task $t = 1 \cdots T$ is to tune hyperparameters associated with a program p_t on a dataset d_t. The program is drawn from a pool of programs $p_t \in \mathbb{P}$ and the dataset belongs to a task-specific set of datasets $d_t \in \mathbb{D}_t$.

The framework of CaTHPO is shown in Figure 1. It contains two major components. (1) A code-aware program representation module (Section 3.1) is trained on $\mathbb{P}$. (2) A cross-program transfer BO module (Section 3.2) relies on the extracted program representations and searches for good hyperparameters for task $t = 1, \cdots, T$.

3.1 Code-aware Program Representation

This module aims to extract a vectorized representation $\mathbf{p}_t$ for each program p_t, from the program's code. Instead of using plain code, we use the Abstract Syntax Tree (AST) obtained by parsing the code. AST has been considered as an easily obtainable, intermediate representation between original source code and a machine- and language-independent description, and has been successfully utilized in many code related tasks (Tang et al. 2022).

Preprocessing. AST is a tree structure where every node has a *type* specifying its meaning (e.g., a function call, a value, etc.). Some nodes (e.g., all leaf nodes) correspond to actual code tokens (they are illustrated as circles in Figure 1) and others correspond to an abstract "part" of the code (they are illustrated as squares in Figure 1) . For parallel computation, we can "flatten" the AST to obtain a sequence of nodes by pre-order traversing the tree (Xie et al. 2021). Furthermore, since ASTs can be very complicated for large programs, to speed up computation and to enhance sample diversity, during training we randomly segment the AST pre-order traversal sequence. Let $\mathbf{o}^i =< \mathbf{o}^i_1, \cdots, \mathbf{o}^i_K >$ denotes a consecutive subsequence of program p_t's AST pre-order traversal sequence $\mathbf{o}^{p_t}$. We also maintain a sequence of binary indicators $\beta^i =< \beta^i_1, \cdots, \beta^i_K >$, where $\beta^i_k = 1$ suggests that the $k-$th node in the training instance corresponds to a code token.

Code Semantic Encoding. We extract a subsequence of code tokens from $\mathbf{o}^i$, denoted as $\mathbf{c}^i =< \mathbf{c}^i_{k_1}, \cdots, \mathbf{c}^i_{k_J} >$ where $\beta^i_{k_j} = 1, \forall 1 \leq j \leq J$. Due to the property of pre-order traversal, code tokens in $\mathbf{c}^i$ are in the same order as in the original source code. We use the standard, word embedding based representation, i.e., $\mathbf{c}^i_{k_j}$ is the embedding vector for the corresponding code token. $\mathbf{c}^i$ is fed into a Transformer encoder, which is based on multi-headed self-attention mechanism. The output $\mathbf{e}^i =< \mathbf{e}^i_{k_1}, \cdots, \mathbf{e}^i_{k_J} >= Transformer(\mathbf{c}^i)$ represents the semantic encoding learned by Transformer in the code contexts.

Code Syntax Encoding. Next, we proceed to integrate the syntactic structure of code. On $\mathbf{o}^i$, for $\beta^i_k = 0$ we use the embedding vector of the node type; for $\beta^i_k = 1$ we replace the node embedding by $\mathbf{e}^i_k$, which is the output of Transformer encoder. In this way, we preserve both learned semantics and syntactic structure. We then input $\mathbf{o}^i$ into a LSTM layer and the output is denoted as $\mathbf{h}^i = LSTM(\mathbf{e}^i)$.

Prediction and Self-supervised Training Task. We use two related self-supervised training tasks, i.e., predict the next AST node type and the next code token in the original AST pre-order traversal sequence. Thus, the training instance is a triplet $(\mathbf{o}^i, q^i, w^i,)$, where q^i is the type of the next AST node, and w^i is the next code token. To make the predictions, the last hidden state of LSTM, i.e., $\mathbf{h}^i_K$, is passed to two seperate MLPs, i.e., MLP^{type} and MLP^{token}, both with softmax output layer. Finally, the code-aware program representation module is optimized to minimize the training loss:

$$\mathbb{L} = \sum_{i \in \mathbb{B}} \left(\mathbb{L}^i_{type} + \mathbb{L}^i_{token} \right), \tag{1}$$

where $\mathbb{L}^i_{type}$ is a cross entropy loss between the predicted AST node type $MLP^{type}(\mathbf{h}^i_K)$ and the ground truth next AST node type q^i. Similarly, $\mathbb{L}^i_{token}$ is the cross entropy loss between the predicted code token $MLP^{token}(\mathbf{h}^i_K)$ and the ground truth next code token w^i. $\mathbb{B}$ is a batch of training instances.

Extracting Program Representations. After the code-aware program representation module is trained, in testing time, for every program p_t, we obtain its full AST preorder traversal sequence $\mathbf{o}^{p_t}$ of length L. $\mathbf{o}^{p_t}$ goes through the trained Transformer and LSTM layers. We apply a mean pooling over the hidden states of LSTM to obtain the program representation, i.e., $\mathbf{p}_t = MeanPooling(\mathbf{h}^{p_t}_1, \mathbf{h}^{p_t}_L)$.

3.2 Cross-program Transfer BO

The goal of hyperparameter tuning is to find the optimal hyperparameter $\mathbf{x}^*$ for $f_t(\mathbf{x})$, where $f_t(\cdot)$ is the hyperparameter performance for task t. Without loss of generality, we can assume that $T-1$ tasks have been tuned (i.e., they are source tasks). We allow each tuning task to be repeated for several runs, but for simplicity here we assume only one run for each task $\mathbb{U}_t$ is recorded, which consists of N observations $\mathbb{U}_t = \{\mathbf{x}_i, y_i\}^N_{i=1}$, where $\mathbf{x}_i$ corresponds to a sampled hyperparameter setting, y_i is a noisy observation, i.e., $y_i = f_t(\mathbf{x}_i) + \epsilon_i, \epsilon_i \sim \mathcal{N}(0, \sigma^2)$. The length of the run N is usually set according to user's budget. As shown in Figure 1, in tuning task T (i.e., the target task), we implement the following steps at each iteration i.[2]

Adjusting Surrogates. As with many previous studies (Wistuba, Schilling, and Schmidt-Thieme 2016; Feurer, Letham, and Bakshy 2018; Wistuba, Schilling, and Schmidt-Thieme 2018), we model each task's hyperparameter performance with an individual GP, $f_t \sim GP_t(\mu_t, k_t)$, where the mean $\mu_t(\cdot)$ and kernel $k_t(\cdot,\cdot)$ can be updated by posterior mean and variance conditioned on observations up to now. Using individual GPs has the advantage of scalability, but fails to capture cross-task correlations. Thus, we next adjust the mean μ_T of the current GP by combining other tasks' GPs:

$$\hat{\mu}_T = \frac{\sum_{t=1}^T w_{T,t}\mu_t}{\sum_{t=1}^T w_{T,t}}. \tag{2}$$

To compute the ensemble weight $w_{T,t}$, we propose a **code sensitive kernel**:

$$w_{T,t} = \alpha_{T,t}\gamma(\mathbb{U}_t, \mathbb{U}_T), \tag{3}$$

where γ is the Epanechnikov quadratic kernel used in TSTR (Wistuba, Schilling, and Schmidt-Thieme 2016), $\alpha_{T,t}$ is the attention score computed on program representations $\mathbf{p}_t, \mathbf{p}_T$,

$$\alpha_{T,t} = \frac{exp(\mathbf{p}_T^T \mathbf{p}_t)}{\sum_{t'=1}^T exp(\mathbf{p}_T^T \mathbf{p}_{t'})}. \tag{4}$$

[2]We omit indicator i in all equations in this subsection. Note that they are computed iteratively at the arrival of each observation $1 \leq i \leq N$.

Since the Epanechnikov quadratic kernel $\gamma(\cdot,\cdot)$ computes the proportions of discordant pairs between tasks, and the attention α computes the distance between pre-trained code feature similarity, the adjusted $\hat{\mu}_T$ strengthens posterior mean of the current GP by similar previous tasks in terms of observation similarity and program similarity.

Following TSTR, we do not combine variance across tasks.

Predicting Candidates. Inspired by MetaBO (Volpp et al. 2020), we present a neural network to serve as Acquisition Function (AF). The AF network is a policy network in Reinforcement Learning . Its output (i.e., action) is to predict the sampling probability over a set of M candidate hyperparameters, i.e., $\mathbb{V} = \{\mathbf{v}_1, \cdots, \mathbf{v}_M\}$. Its input represents the state in reinforcement learning. We make two major improvements over MetaBO.

Firstly, we extend the input of AF network to incorporate the program representation and the adjusted GP mean. Thus, the input of AF is

$$\mathbf{s}_T = \left\{ \mathbf{d}_T, \mathbf{p}_T, \left[\hat{\mu}_T(\mathbf{v}_m), \sigma_T(\mathbf{v}_m), \mathbf{v}_m, m, N \right]_{m=1}^M \right\}, \tag{5}$$

where $\mathbf{d}_T$ represents manually defined metafeatures for the current task's dataset, $\mathbf{p}_T$ represents the pretrained program representations. Incorporating these two terms increases the AF's ability to tailor actions for the current task (which involves a program and a dataset). $\mathbf{v}_m$ is a candidate hyperparameter setting, $\hat{\mu}_T(\mathbf{v}_m)$ is the posterior estimate of mean performance of $\mathbf{v}_m$ augmented by combining similar tasks, $\sigma_T(\mathbf{v}_m)$ is the current posterior estimate of $\mathbf{v}_m$'s performance variance. We concatenate the M candidates to help the AF decide a preference among candidates.

Secondly, we present a **safe controller** to assess the safety of each candidate. The safe controller outputs a binary vector $\mathbf{g}_T \in \mathbb{R}^M$, which has the same dimensionality as the number of candidates,

$$\mathbf{g}_{T,m} = \mathbb{I}(l_T(\mathbf{v}_m) > \tau_T). \tag{6}$$

$\mathbf{g}_{T,m} = 1$ means that $l_T(\mathbf{v}_m)$, the lower bound of candidate m's predicted performance on task T, is better than the performance threshold τ_T. Based on the adjusted mean $\hat{\mu}_T$ and the varianceσ_T, we have

$$l_T(\mathbf{v}_m) = \hat{\mu}_T(\mathbf{v}_m) - \xi\sigma_T(\mathbf{v}_m). \tag{7}$$

The definition of safety threshold τ_T takes into account the performance boundaries from similar tuning tasks,

$$\tau_T = \sum_{t=1}^T \alpha_{T,t} q_t, \tag{8}$$

where q_t is the 25% quartile of observed hyperparameter performances in $\mathbb{U}_t$, α is attention score computed by program representations.

Following MetaBO (Volpp et al. 2020), our AF network also uses an adaptive candidate generation method. It firstly predicts for coarse-grained candidate sets $\mathbb{V}_{coarse}$ spanning the entire domain by sobol sampling. Then, it zooms into a sub-region of the highest scored coarse-grained candidate and randomly selects M candidates around it to form a fine-grained candidate set $\mathbb{V}_{fine}$. The safe controller works as a gate upon the prediction of AF on both coarse-grained and fine-grained candidates. Given the input $\mathbf{s}$, the AF's output is:

$$AF(\mathbf{s}_T) = softmax(\mathbf{g}_T \odot MLP(\mathbf{s}_T)), \tag{9}$$

where $\odot$ is the Hadamard product over two vectors, MLP is a multi-layer perceptron which maps an input vector to a M-dimensional vector.

Sampling, Evaluating, and Updating. Based on the predictions of AF network (augmented by the safe controller), we sample a hyperparameter setting, evaluate it and use the observed evaluation to update the current GP GP_T. The AF network is updated via proximal policy optimization with reward defined as the maximal hyperparameter performance up to the current sample[3].

4 Experiments

We conduct two types of hyperparameter tuning tasks: Spark tuning and Recommender System (RS) tuning, to reflect highly demanded tuning tasks in big data management and AutoML. In Spark tuning, we tune 15 configuration knobs for eight Spark applications on a cluster of eight computing nodes. The hyperparameter performance is measured by Execution time (in seconds). Lower execution time is better. In RS tuning, we tune three hyperparameters for eight recommendation models on a GPU computing server. The hyperparameter performance is measured in NDCG@10, higher NDCG@10 suggests the model achieves higher recommendation accuracy. Each program in Spark and RS tuning is associated with five different open benchmark datasets. Among them, four small datasets are used in source tasks and one large dataset is used in the target task for testing. It simulates the scenario where little tuning overhead (i.e., total time used to train the HPO method) is demanded (Lin et al. 2022), so that small datasets are preferred in meta-training the HPO. Each source task runs once, each target task is repeated for three runs so that our analysis is more reliable. Each run contains 16 samples, i.e., $N = 16$. More details of hyperparameters, programs, and datasets for Spark and RS tuning are provided in the supplementary material. The datasets and codes are available online [4].

4.1 Ablation Study

Here we empirically testify the impact of code-aware cross-program transfer on the surrogate, AF, and safe controller. We use two cases: tuning a Spark application PR and tuning an RS model VAECF. Despite CaTHPO, we consider two other program presentations. (1) codeRand: program representation is a concatenation of an AST encoding using ASTNN (Zhang et al. 2019) and a code encoding using Transformer Encoder (Vaswani et al. 2017). The models are assigned with random network parameters. CodeRand is a "dummy" representation learned from program codes. (2) codeBERT: the program representations are derived using

Surrogates			Spark	RS
Transfer	Task similarity		PR	VAECF
No	N/A		32.41	25.49
Yes	observation		32.89	26.49
	program	codeRand	31.61	26.58
		codeBERT	32.70	27.64
		CaTHPO	**26.79**	**31.03**

Table 1: Tuning performance by variants of surrogates

a pre-trained codeBERT (Feng et al. 2020) model. CodeBERT is a well trained program representation learned from conventional pre-training tasks.

Ablation on surrogates First, we implement variants of the surrogate, while fixing the AF network. (1) We use an individual GP for each tuning task, and there is no transfer across tasks. (2) The current GP is adjusted by a linear combination of source tasks using Epanechnikov quadratic kernel, i.e., Equation 3 without α. This is the method used in the surrogate of TSTR (Wistuba, Schilling, and Schmidt-Thieme 2016). This variant utilizes cross-task transfer based on observation similarity only. (3) The current GP is adjusted by the code sensitive kernel in Equation 3, where the program presentation is codeRand. (4) The current GP is adjusted by the code sensitive kernel in Equation 3, where the program presentation is codeBERT.

We compare CaTHPO with the above variants. The performance (i.e., best hyperparameter performance obtained in the task) are shown in Table 1. We have the following observations. (1) Transfer using only observation similarity does not outperform individual GP without transfer. This observation reveals the importance of an in-depth understanding of task similarities. (2) Program representation by a random network is comparable to program representation by codeBERT. This observation suggests that, existing self-supervised training task is not suitable for learning program representations in hyperparameter tuning. (3) CaTHPO surpasses the variants by a large margin. It reduces PR's execution time by 15%, and increases VAECF's accuracy by 12% than the best variant.

Ablation on AF Next, we implement variants of the AF network, while fixing the surrogates. (1) We use the Expected Improvement (EI), so that there is no transfer across tasks. (2) We use a neural AF, whose input does not include program representation. This variant empowers cross-task transfer, but it does not distinguish programs. (3) We use the input state in Equation 5 for the neural AF, where the program presentation is codeRand. (4) The input state in Equation 5 includes codeBERT based program representations.

Results in Table 2 show that CaTHPO again outperforms different AF variants. There are two discoveries that are different from Table 1. (1) All transfer AFs are significantly better than EI. This suggests that the transfer mechanism is more helpful for AF than for the surrogates. (2) CodeBERT based program representation is better than random program representation. The underlying reason is that care-

[3]For tuning tasks which pursue the smallest performance value, we simply use the negative reward.

[4]https://github.com/XMUDM/CaTHPO

fully designed program representations is crucial to describe the state for the neural AF.

AF		Spark	RS
Transfer	Program representation	PR	VAECF
No	No	50.27	21.74
Yes	No	30.15	28.11
	codeRand	28.58	27.56
	codeBERT	27.92	28.27
	CaTHPO	**26.79**	**31.03**

Table 2: Tuning performance by variants of AFs

Ablation on safe controller Finally, we implement variants of the safe controller, while fixing the surrogates and AF. (1) w/o safe: no safe controller module is implemented. (2) vanilla: a safe controller using a vanilla threshold as described in SafeOPT (Sui et al. 2015). (3) codeRand: the safety threshold is defined by Equation 8, where the attention α is computed based on codeRand. (4) codeBERT: computation of threshold in Equation 8 uses CodeBERT based program representations.

We plot the performance of 16 samples in the tuning tasks in Figure 2. We can see that using a safe controller, regardless of the definition of safe threshold, reduces the performance variance. CaTHPO has the highest average performance and smallest performance variance among samples. This shows that properly identifying similar programs can find more reasonable safety threshold. It is worthy to note that, hyperparameter tuning must balance the exploration-exploitation trade-off. However, the ablation study here shows that, CaTHPO produces a stable exploration of well performing samples, which is advantageous for time efficiency.

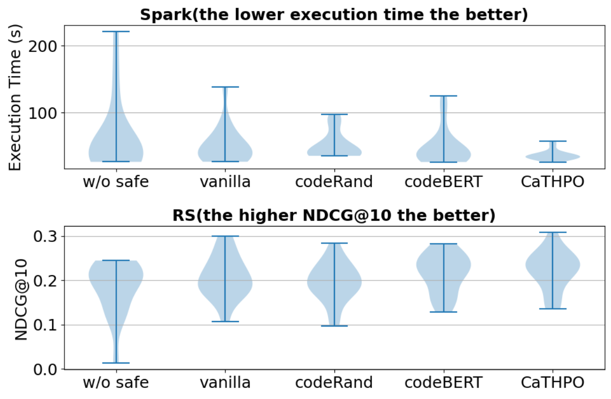

Figure 2: Tuning performance by variants of safe controllers

4.2 Comparative Study

We compare CaTHPO with six state-of-the-art competitors, including (1) standard **BO** with Gaussian Process and Expected Improvement (EI), (2) **ABLR** (Perrone et al. 2018), (3) **TSTR** (Wistuba, Schilling, and Schmidt-Thieme 2016), (4) **FSBO** (Wistuba and Grabocka 2021), (5) **SafeOPT** (Sui et al. 2015) and (6) **MetaBO** (Volpp et al. 2020). We use the public implementations[5] and default settings released by their authors.

Figure 3 reports the tuning performance w.r.t iterations. For each run from iteration $i = 1, \cdots, N$, we first average performance over all tuning tasks, then we plot the average performance of three runs. We have the following observations. (1) The starting point of CaTHPO at iteration $i = 1$ is better than the competitors, which demonstrates CaTHPO's ability to initialize well on a new task by transfer learning from similar tuning tasks. (2) CaTHPO converges faster than the competitors, which shows that CaTHPO achieves satisfying tuning performance with less samples (i.e. sample efficient). (3) At each iteration, CaTHPO generally has smaller variance, which indicates that CaTHPO generates robustly performing samples and its run is likely to end within a shorter time period (i.e., time efficient).

Figure 4 reports the tuning performance w.r.t target tuning tasks (i.e., best performance of a tuning task produced by the hyperparameter samples in each run) by different tuning methods. We observe that CaTHPO steadily outperforms the competitors on all tuning tasks. (1) Figure 4 reports tuning performance of three runs. Hyperparameter performances sampled by the competitors spread in a large range (i.e., a long line in Figure 4) for most tuning tasks. CaTHPO yields similar results (i.e., in most cases they are clustered as a dot in Figure 4) across runs. This indicates that CaTHPO is less likely to be affected by the randomness of each run. (2) CaTHPO produces the best performance averaged over runs for each tuning task. Some tasks, e.g., DT and PR, benefit more from cross program transfer. Compared with the best competitor, CaTHPO reduces the execution time by 25% on DT and 11% on PR. On average, CaTHPO reduces execution time by $24\%, 17\%, 20\%, 14\%, 36\%, 32\%$ of all Spark programs than BO, TSTR, ABLR, MetaBO, FSBO, and SafeOPT. CaTHPO increases NDCG@10 by $22\%, 23\%, 25\%, 10\%, 19\%, 96\%$ of all RS models than BO, TSTR, ABLR, MetaBO, FSBO, and SafeOPT. (3) Comparing non-transfer BO (e.g., BO and SafeOpt) with transfer BOs, we find that transfer BOs do not universally outperform non-transfer BO. Since tuning tasks may be dissimilar, holding a strong assumption and transferring information from all source tasks may not be a good choice.

4.3 Performance on Never-seen Programs

We further investigate the impact of source tuning tasks. The experiment protocol is: we select an RS model as the target tuning task and exclude it from the source tuning tasks. We repeat this procedure for eight RS models. For each target tuning task, there are 28 source tuning tasks (i.e., 4 datasets $\times$ 7 models), while in Section 4.2, there are 32 source tuning tasks. Moreover, the target program is absent in the source tasks, while in Section 4.2 the target program is present in

[5]https://github.com/automl/transfer-hpo-framework

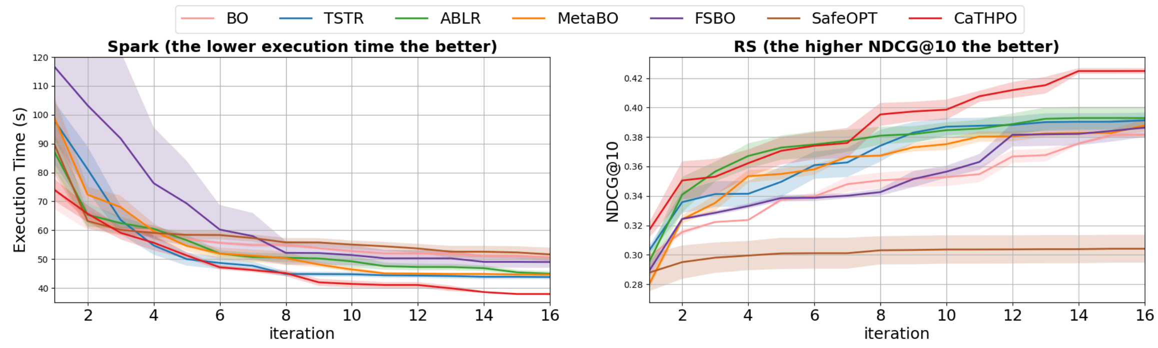

Figure 3: Performance of different tuning methods at each iteration in a run

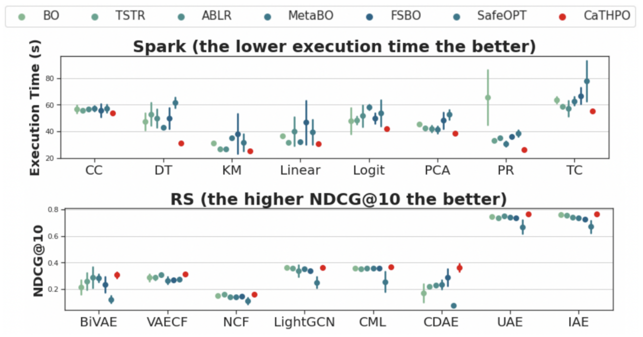

Figure 4: Performance of different tuning methods in each run w.r.t the target tuning task

RS	TSTR	ABLR	MetaBO	FSBO	CaTHPO
BiVAE	15.52	28.1	22.79	22.84	**32.29**
VAECF	25.5	28.41	26.29	25.3	**30.24**
NCF	15.8	15.33	14.64	13.49	**16.86**
LightGCN	35.2	30.57	33.13	33.87	**36.76**
CML	35.13	35.51	34.44	35.3	**36.13**
CDAE	21.5	23.55	20.59	21.05	**32.88**
UAutoRec	73.17	73.49	73.86	73.61	**75.64**
IAutoRec	73.15	74.52	73.07	71.58	**75.61**
Δ	-0.80±1.10	1.50±3.66	-2.48±2.68	-0.80±2.21	**-0.51±0.41**
Spark	**TSTR**	**ABLR**	**MetaBO**	**FSBO**	**CaTHPO**
CC	70.16	65.98	60.17	66.34	**54.98**
DT	68.94	54.22	43.72	63.28	**34.14**
KM	32.22	50.34	35.06	90.19	**25.06**
Linear	32.22	53.3	35.47	76.55	**31.88**
Logit	55.32	53.85	60.32	57.09	**43.28**
PCA	44.88	46.44	42.51	60.13	**39.68**
PR	41.21	48.35	34.68	35.35	**30.24**
TC	68.98	64.79	68.11	75.40	**56.09**
Δ	4.05±3.49	6.60±7.36	3.04±1.77	2.71±2.51	**1.31±1.47**

Table 3: Tuning performance on never-seen programs and the performance change Δ w.r.t warm-start programs

the source tasks. Thus, we have two settings: a setting of **never-seen** program v.s. a setting of **warm-start** program.

We report the tuning performances by CaTHPO and the competitors in Table 3. Note that we do not include BO and SafeOPT as they do not enable transfer learning. Less source tasks and tuning a never seen program will have little impact on BO and SafeOPT. We can see that CaTHPO achieves best results, which verifies that the program representation extraction generalizes well to never-seen programs.

Furthermore, we compare the performance under two settings. We compute the performance change Δ, which is the average performance change over RS models or Spark applications (i.e., never-seen performance minus warm-start performance) $\pm$ the standard deviation. For RS models, negative Δ suggests worse performance on never-seen programs. For Spark applications, positive Δ suggests worse performance on never-seen programs. We can see that almost all tuning methods have some degree of performance degradation. It shows that fewer source tuning tasks has a negative impact on transfer tuning. ABLR has a positive average performance change on RS models, however, the standard deviation is very large. CaTHPO has the smallest Δ, demonstrating that CaTHPO is able to maintain a relatively robust tuning performance with less source tuning tasks and never-seen target program.

5 Conclusion

In this paper we study an under-explored problem: cross-program hyperparameter tuning. We present CaTHPO, the superiority of which is achieved by transferring from similar programs. Program similarities are computed by self-pretrained code-aware program representations. We adopt program similarities to adjust the surrogate, AF, and search space during the whole process of BO and empirically verify the effectiveness of these adjustments. The key idea of CaTHPO, i.e., the code-aware cross-program transfer, has the potential to be adapted to other transfer BOs.

Acknowledgements

Chen Lin is the corresponding author. Chen Lin is supported by the Natural Science Foundation of China (No. 61972328), Alibaba Group through Alibaba Innovative Research program, CCF-Huawei Populus Grove Fund.

References

Anderson, A.; Dubois, S.; Cuesta-Infante, A.; and Veeramachaneni, K. 2017. Sample, Estimate, Tune: Scaling Bayesian Auto-Tuning of Data Science Pipelines. In *DSAA*, 361–372.

Bardenet, R.; Brendel, M.; Kégl, B.; and Sebag, M. 2013. Collaborative hyperparameter tuning. In *ICML*, volume 28 of *JMLR Workshop and Conference Proceedings*, 199–207.

Bergstra, J.; Bardenet, R.; Bengio, Y.; and Kégl, B. 2011. Algorithms for Hyper-Parameter Optimization. In *NeurIPS*, 2546–2554.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP (Findings)*, volume EMNLP 2020 of *Findings of ACL*, 1536–1547.

Feurer, M.; Letham, B.; and Bakshy, E. 2018. Scalable meta-learning for bayesian optimization using ranking-weighted gaussian process ensembles. In *AutoML Workshop at ICML*, volume 7.

Feurer, M.; Springenberg, J. T.; and Hutter, F. 2015. Initializing Bayesian Hyperparameter Optimization via Meta-Learning. In *AAAI*, 1128–1135.

Golovin, D.; Solnik, B.; Moitra, S.; Kochanski, G.; Karro, J.; and Sculley, D. 2017. Google Vizier: A Service for Black-Box Optimization. In *KDD*, 1487–1495. ACM.

He, X.; Zhao, K.; and Chu, X. 2021. AutoML: A survey of the state-of-the-art. *Knowl. Based Syst.*, 212: 106622.

Hsieh, B.-J.; Hsieh, P.-C.; and Liu, X. 2021. Reinforced Few-Shot Acquisition Function Learning for Bayesian Optimization. In Ranzato, M.; Beygelzimer, A.; Dauphin, Y.; Liang, P.; and Vaughan, J. W., eds., *NeurIPS*, volume 34, 7718–7731. Curran Associates, Inc.

Jones, D. R.; Schonlau, M.; and Welch, W. J. 1998. Efficient Global Optimization of Expensive Black-Box Functions. *J. Glob. Optim.*, 13(4): 455–492.

Joy, T. T.; Rana, S.; Gupta, S. K.; and Venkatesh, S. 2016. Flexible Transfer Learning Framework for Bayesian Optimisation. In *PAKDD*, volume 9651 of *Lecture Notes in Computer Science*, 102–114. Springer.

Law, H. C. L.; Zhao, P.; Chan, L. S.; Huang, J.; and Sejdinovic, D. 2019. Hyperparameter Learning via Distributional Transfer. In *NeurIPS*, 6801–6812.

Li, G.; Zhou, X.; and Cao, L. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD*, 2859–2866. ACM.

Lin, C.; Zhuang, J.; Feng, J.; Li, H.; Zhou, X.; and Li, G. 2022. Adaptive Code Learning for Spark Configuration Tuning. In *ICDE*, 1995–2007. IEEE.

Perrone, V.; Jenatton, R.; Seeger, M. W.; and Archambeau, C. 2018. Scalable Hyperparameter Transfer Learning. In *NeurIPS*, 6846–6856.

Perrone, V.; and Shen, H. 2019. Learning search spaces for Bayesian optimization: Another view of hyperparameter transfer learning. In *NeurIPS*, 12751–12761.

Poloczek, M.; Wang, J.; and Frazier, P. I. 2017. Multi-Information Source Optimization. In *NeurIPS*, 4288–4298.

Schilling, N.; Wistuba, M.; and Schmidt-Thieme, L. 2016. Scalable Hyperparameter Optimization with Products of Gaussian Process Experts. In *ECML/PKDD*, volume 9851 of *Lecture Notes in Computer Science*, 33–48. Springer.

Shilton, A.; Gupta, S.; Rana, S.; and Venkatesh, S. 2017. Regret Bounds for Transfer Learning in Bayesian Optimisation. In Singh, A.; and Zhu, J., eds., *AISTATS*, volume 54 of *Proceedings of Machine Learning Research*, 307–315. PMLR.

Snoek, J.; Larochelle, H.; and Adams, R. P. 2012. Practical Bayesian Optimization of Machine Learning Algorithms. In *NeurIPS*, 2960–2968.

Soares, C.; and Brazdil, P. 2000. Zoomed Ranking: Selection of Classification Algorithms Based on Relevant Performance Information. In *PKDD*, volume 1910 of *Lecture Notes in Computer Science*, 126–135. Springer.

Springenberg, J. T.; Klein, A.; Falkner, S.; and Hutter, F. 2016. Bayesian Optimization with Robust Bayesian Neural Networks. In *NeurIPS*, 4134–4142.

Srinivasa, K. G.; and Muppalla, A. K. 2015. *Guide to High Performance Distributed Computing - Case Studies with Hadoop, Scalding and Spark*. Computer Communications and Networks. Springer.

Sui, Y.; Gotovos, A.; Burdick, J. W.; and Krause, A. 2015. Safe Exploration for Optimization with Gaussian Processes. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, 997–1005.

Swersky, K.; Snoek, J.; and Adams, R. P. 2013. Multi-Task Bayesian Optimization. In *NeurIPS*, 2004–2012.

Tang, Z.; Shen, X.; Li, C.; Ge, J.; Huang, L.; Zhu, Z.; and Luo, B. 2022. AST-Trans: Code Summarization with Efficient Tree-Structured Attention. In *ICSE*, 150–162. ACM.

Thornton, C.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *KDD*, 847–855.

Tighineanu, P.; Skubch, K.; Baireuther, P.; Reiss, A.; Berkenkamp, F.; and Vinogradska, J. 2022. Transfer Learning with Gaussian Processes for Bayesian Optimization. In Camps-Valls, G.; Ruiz, F. J. R.; and Valera, I., eds., *AISTATS*, volume 151 of *Proceedings of Machine Learning Research*, 6152–6181. PMLR.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All you Need. In *NIPS*, 5998–6008.

Volpp, M.; Fröhlich, L. P.; Fischer, K.; Doerr, A.; Falkner, S.; Hutter, F.; and Daniel, C. 2020. Meta-Learning Acquisition Functions for Transfer Learning in Bayesian Optimization. In *ICLR*.

Wistuba, M.; and Grabocka, J. 2021. Few-Shot Bayesian Optimization with Deep Kernel Surrogates. In *ICLR*.

Wistuba, M.; Schilling, N.; and Schmidt-Thieme, L. 2015. Learning hyperparameter optimization initializations. In *DSAA*, 1–10. IEEE.

Wistuba, M.; Schilling, N.; and Schmidt-Thieme, L. 2016. Two-Stage Transfer Surrogate Model for Automatic Hyperparameter Optimization. In *ECML/PKDD*, volume 9851 of *Lecture Notes in Computer Science*, 199–214. Springer.

Wistuba, M.; Schilling, N.; and Schmidt-Thieme, L. 2018. Scalable Gaussian process-based transfer surrogates for hyperparameter optimization. *Mach. Learn.*, 107(1): 43–78.

Xie, B.; Su, J.; Ge, Y.; Li, X.; Cui, J.; Yao, J.; and Wang, B. 2021. Improving Tree-Structured Decoder Training for Code Generation via Mutual Learning. In *AAAI*, 14121–14128. AAAI Press.

Yogatama, D.; and Mann, G. 2014. Efficient Transfer Learning Method for Automatic Hyperparameter Tuning. In Kaski, S.; and Corander, J., eds., *AISTATS*, volume 33 of *Proceedings of Machine Learning Research*, 1077–1085. Reykjavik, Iceland: PMLR.

Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; and Liu, X. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*, 783–794.